Eliminación en Arboles de Aproximación Espacial Dinámicos *

Nora Reyes
Departamento de Informática, Universidad Nacional de San Luis.
Ejército de los Andes 950, San Luis, Argentina.
nreyes@unsl.edu.ar

y

Gonzalo Navarro
Departamento de Ciencias de la Computación, Universidad de Chile.
Blanco Encalada 2120, Santiago, Chile.
gnavarro@dcc.uchile.cl

Resumen

El Arbol de Aproximación Espacial (*sa-tree*) es una estructura de datos para búsqueda en espacios métricos recientemente propuesta. Se ha mostrado que tiene buen desempeño comparada contra estructuras de datos alternativas en espacios de alta dimensión o consultas de baja selectividad. La principal desventaja que presentó *sa-tree* fue la de ser una estructura de datos estática, es decir, era dificultoso agregarle o eliminarle nuevos elementos una vez construida. Esto la descartaba para muchas aplicaciones interesantes.

Ya hemos propuesto un buen método para manejar inserciones en el *sa-tree*. En este artículo proponemos y analizamos experimentalmente distintos métodos para realizar eliminaciones. Mostramos que es posible eliminar elementos en *sa-tree*, pagando un bajo costo por permitir total dinamismo y manteniendo aún una buena eficiencia de búsqueda.

Palabras claves: bases de datos, estructuras de datos, algoritmos, espacios métricos.

*Este trabajo ha sido soportado parcialmente por el Proyecto RIBIDI CYTED VII.19 (ambos autores) y por el Centro del Núcleo Milenio para Investigación de la Web, Grant P01-029-F, Mideplan, Chile (segundo autor).

1. Introducción

Actualmente las bases de datos han incluido la capacidad de almacenar nuevos tipos de datos tales como imágenes, sonido, video, etc. Estos tipos de datos son difíciles de estructurar para adecuarlos al concepto tradicional de búsqueda. Así, han surgido aplicaciones en grandes bases de datos en las que se desea buscar objetos similares. Este tipo de búsqueda se conoce con el nombre de *búsqueda aproximada o búsqueda por similitud* y tiene aplicaciones en un amplio número de campos. Algunos ejemplos son bases de datos no tradicionales; búsqueda de texto; recuperación de información; aprendizaje de máquina y clasificación; sólo para nombrar unos pocos.

La necesidad de una respuesta rápida y adecuada, y un uso eficiente de memoria, hace necesaria la existencia de estructuras de datos especializadas que incluyan estos aspectos.

El planteo general del problema es: existe un universo $\mathbb{U}$ de *objetos* y una función de distancia positiva d: $\mathbb{U} \times \mathbb{U} \to \mathbb{R}^+$ definida entre ellos. Esta función de distancia satisface los tres axiomas que hacen que el conjunto sea un *espacio métrico*: positividad estricta ($d(x, y) = 0 \Leftrightarrow x = y$), simetría ($d(x, y) = d(y, x)$) y desigualdad triangular ($d(x, z) \leq d(x, y) + d(y, z)$). Mientras más "similares" sean dos objetos menor será la distancia entre ellos. Tenemos una *base de datos* finita $S \subseteq \mathbb{U}$, que es un subconjunto del universo de objetos y puede ser preprocesada (v.g. para construir un índice). Luego, dado un nuevo objeto del universo (un *query* q), debemos recuperar todos los elementos similares que se encuentran en la base de datos. Existen dos consultas típicas de este tipo:

Búsqueda por rango: recuperar todos los elementos de S que están a distancia r de un elemento q dado.

Búsqueda de k vecinos más cercanos: dado q, recuperar los k elementos más cercanos a q en S.

La distancia se considera costosa de evaluar (pensar, por ejemplo, en comparar dos huellas dactilares). Así, es usual definir la complejidad de la búsqueda como el número de evaluaciones de distancia realizadas, dejando de lado otras componentes tales como tiempo de CPU para computaciones colaterales, y aún tiempo de E/S. Dada una base de datos de $|S| = n$ objetos el objetivo es estructurar la base de datos de forma tal de realizar menos de n evaluaciones de distancia (trivialmente n bastarían).

Un caso particular de este problema surge cuando el espacio es un conjunto D-dimensional de puntos y la función de distancia pertenece a la familia L_p de Minkowski: $L_p = (\sum_{1 \leq i \leq d} |x_i - y_i|^p)^{1/p}$. Los casos especiales más conocidos son $p = 1$ (distancia de Manhattan), $p = 2$ (distancia Euclídea) y $p = \infty$ (distancia máxima), es decir, $L_\infty = \max_{1 \leq i \leq d} |x_i - y_i|$

Existen métodos efectivos para buscar sobre espacios D-dimensionales, tales como kd-trees [1] o R-trees [4]. Sin embargo, para 20 dimensiones o más esas estructuras dejan de trabajar bien. Nos dedicamos en este artículo a espacios métricos generales, aunque las soluciones son también adecuadas para espacios D-dimensionales. Es interesante notar que el concepto de "dimensionalidad" se puede también traducir a espacios métricos: la característica típica en espacios de alta dimensión con distancias L_p es que la distribución de probabilidad de las distancias tiene un histograma concentrado, haciendo así que el trabajo realizado por cualquier algoritmo de búsqueda por similaridad sea más dificultoso [2, 3].

Para espacios métricos generales existen numerosos métodos para preprocesar la base de datos con el fin de reducir el número de evaluaciones de distancia [3]. Todas aquellas estructuras trabajan básicamente descartando elementos mediante la desigualdad triangular, y la mayoría usa la técnica dividir para conquistar.

El Árbol de Aproximación Espacial (*sa-tree*) es una estructura de esta clase propuesta recientemente [5, 8], que se basa sobre un nuevo concepto: más que dividir el espacio de búsqueda, aproximarse al query espacialmente. Además de ser algorítmicamente interesante por sí mismo, se ha mostrado que *sa-tree* da un mejor balance espacio-tiempo que las otras estructuras existentes sobre espacios métricos de alta dimensión o consultas con baja selectividad, lo cual ocurre en muchas aplicaciones.

El *sa-tree*, sin embargo, tiene algunas debilidades importantes. La primera es que es relativamente costoso de construir en bajas dimensiones, comparado con otros índices. La segunda es que en bajas dimensiones o para consultas con alta selectividad (r o k pequeños), el desempeño de su búsqueda es pobre respecto de alternativas simples. La tercera es que es una estructura de datos estática: una vez construida, es difícil agregarle o eliminarle elementos a ella. Estas debilidades hacen al *sa-tree* inadecuado para aplicaciones importantes tales como bases de datos multimedia.

El dinamismo completo no es común en estructuras de datos métricas [3]. Aunque es bastante usual permitir inserciones eficientes, no ocurre lo mismo con las eliminaciones. En varios índices uno puede eliminar algunos elementos, pero existen determinados elementos que nunca pueden ser eliminados. Esto es particularmente problemático en el escenario de los espacios métricos, donde los objetos podrían ser muy grandes (v.g. imágenes) y sería indispensable eliminarlos físicamente. Nuestros algoritmos permiten eliminar cualquier elemento de un *sa-tree*, lo cual es destacable sobre una estructura de datos cuya concepción original fue marcadamente estática [5].

Además de los logros anteriores, ya encontramos cómo obtener grandes mejoras en tiempo de cons-trucción y búsqueda para espacios de baja dimensión o consultas de alta selectividad. El método consiste en limitar la aridad del árbol e involucra nuevos algoritmos sobre esta estructura de datos. A menor aridad el árbol es más barato de construir. Sin embargo, en tiempo de búsqueda, la mejor aridad depende de la dimensión del espacio y de la selectividad de la consulta. En particular, para dimensiones bajas, obtenemos mejores tiempos de construcción y de búsqueda simultáneamente.

El resultado final es una estructura de datos que puede ser útil en una amplia variedad de aplicaciones. Esperamos que el *sa-tree* dinámico reemplace a la versión estática en los desarrollos venideros. Ya obtuvimos un buen método para soportar sólo inserciones sobre el *sa-tree*[7], que además posee un parámetro que le permite adaptarse mejor a diferentes dimensiones. El *sa-tree* original se adaptaba a la dimensión pero no óptimamente. Ahora mostramos en detalle que también se pueden realizar eliminaciones obteniendo así una estructura totalmente dinámica.

Para los experimentos de este artículo hemos seleccionado dos espacios métricos. El primero es un diccionario de Inglés de 69069 palabras. La distancia es la distancia de edición, es decir, el mínimo número de inserciones, eliminaciones o reemplazos de caracteres que deben hacerse para igualar las palabras. El segundo espacio es un espacio de 100000 vectores aleatoriamente generados con una distribución de Gauss con media 1 y varianza 0.1; de dimensión 101, distribuidos en 200 clusters y utilizando distancia Euclídea. Consideramos que ambos espacios sirven para ilustrar el comportamiento del *sa-tree*.

En los todos los casos, construimos los índices con a lo sumo el 90 % de los puntos y usamos siempre el 10 % restante (aleatoriamente elegido) como queries. Para los experimentos con eliminaciones, en un índice de n elementos, seleccionamos una fracción aleatoria de ellos y los eliminamos del índice. Los resultados sobre estos dos espacios son representativos de otros varios espacios métricos testeados: imágenes de la NASA, diccionarios en otros lenguajes, otras dimensiones, etc.

2. Árbol de Aproximación Espacial

Para mostrar la idea general de la aproximación espacial se utilizan las *búsquedas del vecino más cercano* (aunque después veremos cómo resolver todos los tipos de consultas).

En este modelo, dado un punto $q \in \mathbb{U}$ y estando posicionado en algún elemento $a \in S$ el objetivo es moverse a otro elemento de S que esté más cerca "espacialmente" de q que a. Cuando no es posible realizar más este movimiento, se ha alcanzado el elemento más cercano a q de S.

Estas aproximaciones son efectuadas sólo vía los "vecinos". Cada elemento $a \in S$ tiene un conjunto de vecinos $N(a)$. La estructura natural para representar esto es un grafo dirigido. Los nodos son los elementos del conjunto y los elementos vecinos son conectados por un arco. La búsqueda procede sobre tal grafo simplemente posicionándose sobre un nodo arbitrario a y considerando todos sus vecinos. Si ningún nodo está más cerca de q que a, entonces se responde a a como el vecino más cercano a q. En otro caso, se selecciona algún vecino b de a que está más cerca de q que a y se mueve a b. Para obtener el *sa-tree* se simplifica la idea general comenzando la búsqueda en un nodo fijo y así realmente se obtiene un árbol. El *sa-tree* está definido recursivamente; la propiedad que cumple la raíz a (y a su vez cada uno de los siguientes nodos) es que los hijos están más cerca de la raíz que de cualquier otro punto de S. En otras palabras: $\forall x \in S, x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\}, d(x, y) > d(x, a)$.

Finalmente, el radio de cobertura $R(a)$ se usa para podar la búsqueda, no entrando en los subárboles tales que $d(q, a) > R(a) + r$, porque ellos no pueden contener elementos útiles.

Por falta de espacio se refiere al lector al artículo sobre *sa-tree* original [5, 8] para más detalles sobre esta versión de la estructura.

2.1. Versión dinámica

Hemos considerado previamente varias alternativas de inserción en [6, 8]. Allí los mejores métodos resultaron los llamados "timestamping" e "inserción en la fringe". Luego en [7] propusimos una nueva técnica basada sobre ideas de estos dos métodos. En estos trabajos previos sólo se analizaron inserciones y no eliminaciones.

Timestamping permitía insertar un elemento con una técnica muy similar a la versión estática, registrando el tiempo en que se insertó cada elemento. Esta técnica tuvo un desempeño muy similar al de la versión estática y evitaba la reconstrucción. La inserción en la fringe, en el otro sentido, limitaba el tamaño máximo del árbol en el cual se podía insertar un nuevo elemento, con la idea de reconstruir sólo pequeños subárboles. La técnica nos permite no insertar en el punto en que la versión estática requeriría hacerlo, y sí hacerlo en un punto más bajo del árbol. Sorpresivamente, esta técnica mejoró el desempeño en dimensiones bajas.

Siguiendo esta línea determinamos que el hecho clave es la reducción en la aridad de estos árboles. Más aún la principal razón del pobre desempeño de *sa-tree* en espacios de baja dimensión es su aridad excesivamente alta (el árbol adapta su aridad automáticamente a la dimensión, pero no lo hace óptimamente). Por lo tanto , nos concentramos directamente sobre la máxima aridad permitida y la transformamos en un parámetro para sintonizar. La misma técnica que usábamos para limitar el tamaño del árbol a reconstruir la utilizamos para limitar la aridad del árbol. Más aún, mezclamos esta técnica con timestamping y no neceistamos así compensar el costo de reconstrucción, y de esta manera obtuvimos lo mejor de ambas soluciones.

Observar que uno de los aspectos agradables del *sa-tree* original era que no tenía ningún parámetro, y así cualquier inexperto podía usarlo. Nuestro nuevo parámetro no es perjudicial en este sentido, ya

que puede ser tomado como ∞ para obtener el mismo desempeño que *sa-tree* original. En otro sentido, obtuvimos grandes mejoras en dimensiones bajas tomando adecuadamente la aridad máxima del árbol. Para más detalles ver [7].

Para construir incrementalmente el *sa-tree* fijamos una aridad máxima para el árbol y también guardamos un timestamp del tiempo de inserción de cada elemento. Cuando insertamos un nuevo elemento x, lo agregamos como vecino en el punto más apropiado sólo si la aridad del nodo aún no es la máxima. En otro caso, a pesar de que x está más cerca de ese nodo que de cualquiera de sus vecinos, forzamos a que x elija el vecino más cercano del nodo corriente y bajamos en el árbol hasta que alcancemos un nodo, tal que se satisfaga que x esté más cerca de él que de cualquiera de sus vecinos y además que su aridad no sea maximal (esto eventualmente ocurre en una hoja del árbol). En este punto agregamos a x al final de la lista de los vecinos, le colocamos el timestamp corriente a x e incrementamos el timestamp corriente. La Figura 1 ilustra el proceso de inserción. La función se invoca como `Insertar(`a`,`x`)`, donde a es la raíz del árbol y x es el elemento a insertar.

Insertar `(Nodo` a`, Elemento` x`)`
1. $R(a) \leftarrow \max(R(a), d(a,x))$, $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
2. `If` $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ `Then`
3. $N(a) \leftarrow N(a) \cup \{x\}$, $N(x) \leftarrow \emptyset$
4. $R(x) \leftarrow 0$, $time(x) \leftarrow CurrentTime$
5. `Else Insertar (`c`,`x`)`

Figura 1: Inserción de un nuevo elemento x en un *sa-tree* dinámico con raíz a. $MaxArity$ es la máxima aridad del árbol y $CurrentTime$ es el tiempo corriente, que se incrementa luego de cada inserción.

Notar que si se visitan los vecinos de izquierda a derecha tenemos los timestamp en orden creciente y también que el timestamp del padre es siempre más viejo que el de sus hijos. Pero, ahora no estamos seguros que el nuevo elemento x se haya insertado como vecino del primer nodo a más cercano a x que a su vecindad $N(a)$ en su paso. Es posible que la aridad de a ya fuera la máxima y que hayamos forzado a x a elegir un vecino de a. Consideraremos pronto cómo esto afecta el proceso de búsqueda.

El *sa-tree* ahora se puede construir comenzando con un único nodo a donde $N(a) = \emptyset$ y $R(a) = 0$, y luego realizar sucesivas inserciones. En [7] se puede observar en detalle el proceso de inserción y comparaciones entre el costo de construcción incremental usando nuestra técnica contra la construcción estática. Allí se muestran diferentes aridades y en todos los casos la construcción mejora a medida que reducimos la aridad del árbol, siendo mucho mejor que la construcción estática. Por lo tanto, esto que la aridad reducida es un factor clave para bajar los costos de construcción. Como un ejemplo podemos mencionar que la construcción incremental en el espacio de vectores de dimensión 15 cuesta al menos un 50 % menos que la estática, en el espacio de vectores de Gauss cuesta cerca de un 80 % menos y en el de strings cuesta cerca de un 8 % más. El costo de inserción con aridad A es $A \log_A n$. Sobre aridad ilimitada la aridad promedio es $A = O(\log n)$, así el costo de construcción por elemento es $O(\log^2 n / \log \log n)$ [8]. Ahora consideramos cómo una aridad reducida afecta el tiempo de búsqueda.

2.1.1. Búsquedas

Cuando buscamos debemos considerar dos hechos. El primero es que, cuando se insertó un elemento x, se pudo no haber elegido a un nodo a en su paso como su padre porque su aridad ya era

maximal. Así en lugar de elegir el más cercano a x entre $\{a\} \cup N(a)$, pudimos haber tenido que elegir sólo entre $N(a)$. Esto significa no tenemos que considerar a a en la minimización. El segundo hecho es considerar que, cuando se insertó a x, no se encontraban en el árbol elementos con timestamp más alto, así x podría elegir su vecino más cercano sólo entre aquellos elementos más viejos que él.

Por lo tanto, consideramos los vecinos $\{b_1, \ldots, b_k\}$ de a desde el más viejo al más nuevo, sin tener en cuenta a a, y realizamos la minimización a medida que atravesamos la lista. Esto significa que vamos a entrar en el subárbol de b_i si $d(q, b_i) \leq \min(d(q, b_1), \ldots, d(q, b_{i-1})) + 2r$. Repetimos de nuevo la razón: entre la inserción de b_i y b_{i+j} pudieron haber aparecido nuevos elementos que elegieron a b_i sólo porque b_{i+j} no estaba presente aún, así podemos perder un elemento si no entramos en b_i porque ahora existe b_{i+j}.

Podemos usar mejor la información del timestamp para reducir el trabajo a realizar en los vecinos más viejos. Digamos que $d(q, b_i) > d(q, b_{i+j}) + 2r$. Tenemos que entrar en el subárbol de b_i siempre, porque b_i es más viejo. Sin embargo, sólo los elementos con timestamp más pequeño que el de b_{i+j} se deberían considerar cuando busquemos dentro de b_i; los elementos más jóvenes ya han visto a b_{i+j} y ellos no pueden ser interesantes para la búsqueda si están dentro de b_i. Como los nodos padres son más viejos que sus descendientes, tan pronto como encontremos un nodo dentro del subárbol de b_i con timestamp más grande que el de b_{i+j} podemos parar la búsqueda en esa rama, porque todo su subárbol será aún más joven.

La Figura 2 muestra el algoritmo de búsqueda, el cual se invoca como `BúsquedaRango(`a`,`q`,`r`,`∞`)` inicialmente, donde a es la raíz del árbol. Notar que siempre se conoce $d(a, q)$, excepto en la primera invocación. A pesar de la naturaleza cuadrática del loop implícito en las líneas 4 y 6, el query se compara sólo una vez contra cada vecino.

```
BúsquedaRango (Nodo a, Query q, Radio r, Timestamp t)
1. If time(a) < t  ∧  d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Informar a
3.    d_min  ←  ∞
4.    For b_i ∈ N(a) en orden creciente de timestamp Do
5.       If d(b_i,q) ≤ d_min + 2r Then
6.          k ← mín  {j > i,  d(b_i,q) > d(b_j,q) + 2r}
7.          BúsquedaRango (b_i,q,r,time(b_k))
8.       d_min ← mín{d_min, d(b_i,q)}
```

Figura 2: Buscando q con radio r en un *sa-tree* dinámico.

La Figura 3 compara esta técnica contra la estática. En el caso de los strings, el método estático da un tiempo de búsqueda levemente mejor comparado con la técnica dinámica. En el caso de los vectores de dimensión 101 con distribución de Gauss al construir el *sa-tree* dinámico con cualquiera de las aridades elegidas superamos ampliamente a la versión estática. Los experimentos en este espacio y otros de baja dimensión muestran que pequeñas aridades exhiben mejoras significativas sobre el tiempo de búsqueda del método estático. La mejor aridad para la búsqueda depende del espacio métrico, pero la regla general es que aridades bajas son buenas para dimensiones bajas o pequeños radios de búsqueda.

Consideramos el número de evaluaciones de distancia en lugar de tiempo de CPU, porque el costo de CPU sobre el número de evaluaciones de distancia es despreciable en el *sa-tree*, a diferencia de otras estructuras.

Es importante notar que hemos obtenido dinamismo y también mejorado el desempeño de la construcción. En algunos casos también hemos mejorado ampliamente las búsquedas, mientras que en otros casos hemos pagado un pequeño precio por tener dinamismo. En general, esta estructura se vuelve una elección muy conveniente. Esta técnica se adapta fácilmente para las búsquedas del vecino más cercano con los mismos resultados. Se realizan las búsquedas de vecino más cercano simulando una búsqueda por rango y se va reduciendo el radio de búsqueda a medida que se desarrolla. Omitimos detalles por falta de espacio.

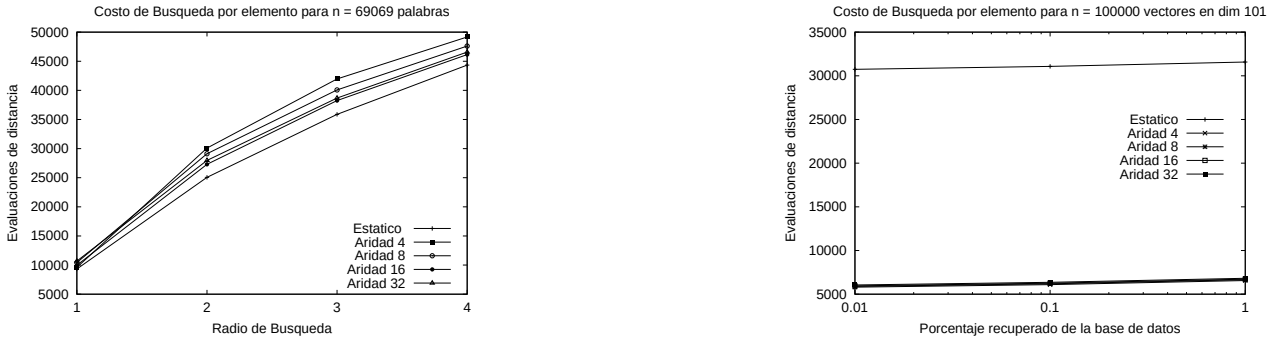

Figura 3: Costos de búsqueda Estática versus Dinámica.

3. Eliminaciones

Para eliminar un elemento x, el primer paso es localizarlo en el árbol. A diferencia de la mayoría de las estructuras de datos clásicas, hacer esto no es equivalente a simular la inserción de x y ver a donde nos guía en el árbol. La razón es que el árbol era diferente cuando se insertó a x. Si x se insertara de nuevo, podría elegir ir por un paso diferente en el árbol, el cual no existía al momento de la primera inserción.

Una solución elegante a este problema es realizar una búsqueda por rango con radio cero, es decir, una consulta de la forma $(x, 0)$. Esto es razonablemente barato y nos llevará por todos los lugares en el árbol donde se podría haber sido insertado a x.

En otro sentido, es dependiente de la aplicación si esta búsqueda es o no necesaria. La aplicación podría retornar información cuando se inserta un objeto en la base de datos. Esta información podría contener un puntero al correspondiente nodo del árbol, y agregando en el árbol punteros al padre permitiríamos ubicar el paso sin costo adicional (en términos de evaluaciones de distancia). Así, en lo que sigue, no consideramos la localización del objeto como parte del problema de eliminación, aunque hemos mostrado cómo hacerlo si es necesario.

Hemos analizado varias alternativas para eliminar elementos desde un *sa-tree* dinámico. Desde el comienzo hemos descartado la opción trivial de marcar al elemento como eliminado sin eliminarlo realmente. Como se explicó, esta solución probablemente sea inaceptable en la mayoría de las aplicaciones. Asumimos que el elemento tiene que ser eliminado físicamente. Podemos, si se desea, mantener su nodo en el árbol, pero no el objeto mismo.

Claramente una hoja del árbol siempre se puede remover sin ninguna complicación, así que nos ocuparemos en cómo remover nodos internos del árbol.

3.1. Nodos Ficticios

Nuestra primera alternativa para eliminar un elemento x es dejar su nodo en el árbol (sin contenido) y marcarlo como eliminado. Llamamos a estos nodos *ficticios*. Aunque es poco costoso y

simple al momento de la eliminación, debemos ver ahora cómo llevar a cabo una búsqueda consistente cuando algunos nodos no contienen objetos.

Básicamente, si el nodo $b \in N(a)$ es ficticio, no tenemos suficiente información para poder evitar entrar en el subárbol de b cuando hemos alcanzado a a. Así no podemos incluir a b en la minimización y debemos entrar siempre a su subárbol (excepto si podemos usar el timestamp de b para podar la búsqueda).

La búsqueda realizada en tiempo de inserción, en otro sentido, tiene que seguir sólo un paso en el árbol. En este caso, se es libre de elegir insertar el nuevo elemento en cualquier vecino ficticio del nodo corriente, o en el vecino más cercano no ficticio. Sin embargo, una buena política es tratar de no incrementar el tamaño de los subárboles cuya raíz es un nodo ficticio, porque eventualemente ellos se deberán reconstruir (ver más adelante).

Así, aunque la eliminación es simple, se degrada el proceso de búsqueda.

3.2. Reinsertando Subárboles

Una idea muy difundida en la comunidad de búsqueda por rango Euclídea es que reinsertar los elementos de una página de disco es beneficioso porque, con más elementos en el árbol, el espacio puede agruparse mejor. Seguimos este principio ahora para obtener un método con eliminaciones más costosas, pero con buen desempeño de búsqueda.

Cuando se elimina un nodo x, desconectamos el subárbol con raíz x desde el árbol principal. Esta operación no afecta la correctitud del árbol restante, pero ahora tenemos que reinsertar los subárboles con raíces en los nodos de $N(x)$. Para hacerlo eficientemente tratamos de reinsertar subárboles completos, siempre que sea posible.

Para reinsertar un subárbol con raíz y, seguimos los mismos pasos que si insertáramos un nuevo objeto y y encontramos el punto de inserción a. La diferencia es que asumimos que y es un objeto "gordo" con radio $R(y)$. Es decir, podemos elegir poner el subárbol completo con raíz y como un nuevo vecino de a sólo si $d(y, a) + R(y)$ es menor que $d(y, b)$ para cualquier $b \in N(a)$. Similarmente, podemos elegir bajar por el vecino $c \in N(a)$ slo si $d(y, c) + R(y)$ es menor que $d(y, b)$ para cualquier $b \in N(a)$. Cuando ninguna de estas condiciones se cumple, estamos forzados a dividir el subárbol con raíz y en sus elementos: uno es el elemento y y los otros son los subárboles con raíces $N(y)$. Una vez dividido el subárbol, continuamos el proceso de inserción con cada uno de los componentes por separado.

Cada vez que insertamos un nodo o un subárbol, obtenemos un timestamp nuevo para el nodo o la raíz del subárbol. Los elementos dentro del subárbol obtendrán sus nuevos timestamps manteniendo el ordenamiento relativo dentro de los elementos del subárbol. La manera más sencilla de hacerlo es asumir que los timestamps se almacenan relativos a los de su padre. De este modo, no se tiene que hacer nada. Necesitamos, sin embargo, almacenar en cada nodo el diferencial máximo de tiempo almacenado en el subárbol, para actualizar adecuadamente a $CurrentTime$ cuando se reinserta un subárbol completo. En tiempo de inserción esto se hace fácilmente y por simplicidad se omite en el pseudocódigo.

Durante la reinserción, también modificamos los radios de cobertura de los nodos a del árbol atravesados. Cuando insertamos un subárbol completo tenemos que sumar $d(y, a) + R(y)$, lo que es mayor que lo necesario. Esto involucra un costo en tiempo de búsqueda por haber reinsertado un subárbol completo de una sola vez.

Notar que puede parecer que, cuando buscamos el lugar para reinsertar los subárboles de un nodo x eliminado, uno podría ahorrar tiempo comenzando la búsqueda en el padre de x; sin embargo, el

árbol ha cambiado desde el momento en que se creó el subárbol de x, y ahora pueden existir nuevas elecciones.

La Figura 4 muestra el algoritmo para reinsertar un árbol con raíz y en un *sa-tree* dinámico con raíz a. La eliminación de un nodo x se hace primero localizándolo en el árbol (digamos, $x \in N(b)$), luego removiéndolo desde $N(b)$, y finalmente reinsertando cada subárbol $y \in N(x)$ usando `ReinsertarT(`a`,`y`)`.

```
ReinsertarT (Nodo a, Nodo y)
1.  If |N(a)| < MaxArity Then M ← {a} ∪ N(a)  Else M ← N(a)
2.  c1 ← argmin_{b∈M} d(b, y)
3.  c2 ← argmin_{b∈M−{c1}} d(b, y)
4.  If d(c1, y) + R(y) ≤ d(c2, y) Then // mantener junto el subárbol
5.     R(a) ← máx(R(a), d(a, y) + R(y))
6.     If c1 = a Then // insertarlo aquí
7.        N(a) ← N(a) ∪ {y}
8.        time(y) ← CurrentTime // el subárbol se corre automáticamente
9.     Else ReinsertarT (c1, y) // bajar
10. Else // dividir el subárbol
11.    For z ∈ N(y) Do ReinsertarT (a, z)
12.    N(y) ← ∅, R(y) ← 0
13.    ReinsertarT (a, y)
```

Figura 4: Algoritmo simple para reinsertar un subárbol con raíz y en un *sa-tree* dinámico con raíz a.

3.3. Reinsertando Subárboles de a Elemento

La reinserción de subárboles completos acarrea, en algunos espacios, un inconveniente adicional que puede degradar el desempeño de la búsqueda. Si se debe reinsertar un subárbol con raíz y, el proceso atraviesa un paso desde la raíz del árbol hasta llegar a un punto en el que o reinserta el subárbol completo o debe dividirlo, y reinsertar a y y a cada uno de los subárboles con raíces en $N(y)$. En cada uno de los nodos a que atraviesa en ese paso actualiza, inevitablemente, el $R(a)$ a un valor posiblemente mayor que el necesario ($d(a, y) + R(y)$). Esto provoca que dichos radios de cobertura puedan quedar sobredimensionados (aún si no se logra reinsertar completo al subárbol).

Así una alternativa, similar a la anterior, que no provoca sobredimensión en los radios de cobertura es reinsertar de a uno los elementos del subárbol con raíz y. En este caso la búsqueda tiene un mejor desempeño, aunque no así la eliminación. El algoritmo para realizar la reinserción elemento a elemento se consigue variando el algoritmo de Figura 4. El proceso de eliminación en este caso es el mismo que ya se describió anteriormente.

3.4. Optimización.

Otra optimización al proceso de reinserción de subárboles utiliza más inteligentemente los timestamps. Supongamos que x será eliminado, y sea $A(x)$ el conjunto de los ancestros de x, es decir, todos los nodos en el paso desde la raíz a x. Para cada nodo c que pertenece al subárbol con raíz x tenemos que $A(x) \subset A(c)$. Así, cuando se insertó el nodo c, se comparó contra todos los vecinos de cada nodo en $A(x)$ cuyo timestamp sea menor que el de c. Usando esta información podemos evitar evaluar las distancias a esos nodos cuando los revisitamos al momento de reinsertar c. Es decir, cuando buscamos el vecino más cercano a c, sabemos que el que está en $A(x)$ está más cerca de

c que cualquier otro vecino más viejo, así tenemos que considerar sólo los más nuevos. Notar que esto es válido en la medida que reentremos por el mismo paso donde fue insertado previamente. Esta optimización también es aplicable al método de reinserción de a elementos.

El costo promedio de la reinserción de subárbol es como sigue. Asumimos que reinsertamos los elementos de a uno, que el árbol tiene siempre aridad A y que está perfectamente balanceado. Entonces, el tamaño promedio de un subárbol aleatoriamente elegido se vuelve $\log_A n$. Como cada una de las (re)inserciones cuesta $A \log_A n$, el costo promedio de eliminación es $A \log_A^2 n$. Esto es mucho más costoso que una inserción.

4. Combinando Métodos

Tenemos tres métodos. Nodos ficticios elimina elementos sin costo pero degrada el desempeño de la búsqueda del árbol. Reinserción de subárboles y de a elementos hacen una reinserción de subárbol costosa pero tratan de mantener la calidad de la búsqueda del árbol. Observar que el costo de reinsertar un subárbol no sería muy diferente si él contiene nodos ficticios. Así, podríamos remover todos los nodos ficticios con una única reinserción de subárbol y por lo tanto amortizar el alto costo de la reinserción sobre muchas eliminaciones.

Nuestra idea es asegurar que cada subárbol tenga a lo sumo una fracción α de nodos ficticios. Decimos que tales subárboles son "balanceados". Cuando marcamos un nuevo nodo como ficticio, verificamos si no tenemos un desbalanceo. En ese caso, se descarta x y se reinsertan sus subárboles. La única diferencia es que nunca reinsertamos un subárbol cuya raíz sea un nodo ficticio, sino que descomponemos el subárbol y descartamos la raíz ficticia.

Una complicación surge porque remover el subárbol con raíz x puede desbalancear varios ancestros de x, aún si x fuera sólo una hoja que puede removerse directamente, y aún si el ancestro no tiene como raíz un nodo ficticio.

Entonces, optamos por una solución simple. Buscamos el ancestro más bajo b de x que se haya desbalanceado y reinsertamos todo el subárbol con raíz b. Por esta complicación, reinsertamos subárboles completos (o elemento a elemento) sólo cuando ellos no contengan nodos ficticios.

Esta técnica tiene un buen desempeño. Aún si reinsertamos los elementos uno por uno (en lugar de subárboles completos) y tendríamos la garantía que reinsertaríamos un subárbol sólo cuando una fracción α de sus elementos son ficticios. Esto significaría que si m fuera el tamaño del subárbol a reconstruir, pagaríamos $m(1-\alpha)$ reinserciones por cada eliminación hecha en el subárbol. De aquí, el costo amortizado de una eliminación sería a lo sumo $(1-\alpha)/\alpha$ veces el costo de una inserción, es decir, $(1-\alpha)/\alpha \; A \log_A n$. Asintóticamente, el árbol trabajaría como si permanentemente tuviera una fracción α de nodos ficticios. Por lo tanto, podemos controlar el balance entre costos de búsqueda y de eliminación. Notar que nodos ficticios puros se corresponde a $\alpha = 1$ y reinserción de subárboles pura (o de a elementos pura) corresponde a $\alpha = 0$.

4.1. Comparación Experimental

Comparemos ahora los tres métodos para realizar eliminaciones sobre el espacio de palabras usando aridad 16. La Figura 5 muestra el costo para el primer 10 % (izquierda) o 40 % (derecha) de la base de datos. Sobre la izquierda mostramos el caso de reinserción completa (es decir, reinsertando los subárboles después de cada eliminacioñ), con y sin la optimización final propuesta. Como se puede observar, ahorramos cerca del 50 % de los costos de eliminación con la optimización. También mostramos que raramente se pueden reinsertar subárboles completos, porque reinsertar los elementos de a uno tiene el mismo costo. Así se podría usar el algoritmo simplificado sin sacrificar mucho.

También mostramos el método combinado con $\alpha = 1\,\%$, $3\,\%$ and $5\,\%$. A la derecha mostramos valores mayores de α, desde $0\,\%$ (reinsercíon completa) hasta $100\,\%$ (nodos ficticios puros), como así también porcentajes más grandes de eliminaciones (sólo usaremos de ahora en más la versión optimizada de reinserciones).

Comparamos los métodos eliminando diferentes porcentajes de la base de datos para que se aprecie no sólo el costo de eliminación por elemento sino también para mostrar el efecto acumulativo de las eliminaciones sobre la estructura (por la sobredimensión de los radios de cobertura).

Se puede observar que, aún con reinserción completa, el costo de eliminación individual no es tan alto. Por ejemplo, el costo de inserción promedio en este espacio es cercano a 58 evaluaciones de distancia por elemento. Con el método optimizado cada eliminación cuesta cerca de 173 evaluaciones de distancia, es decir, 3 veces el costo de una inserción. El método combinado mejora sobre este: usando α tan pequeño como $1\,\%$ tenemos un costo de eliminación de 65 evaluaciones de distancia, que se acerca al costo de las inserciones, y con $\alpha = 3\,\%$ éste se reduce a 35.

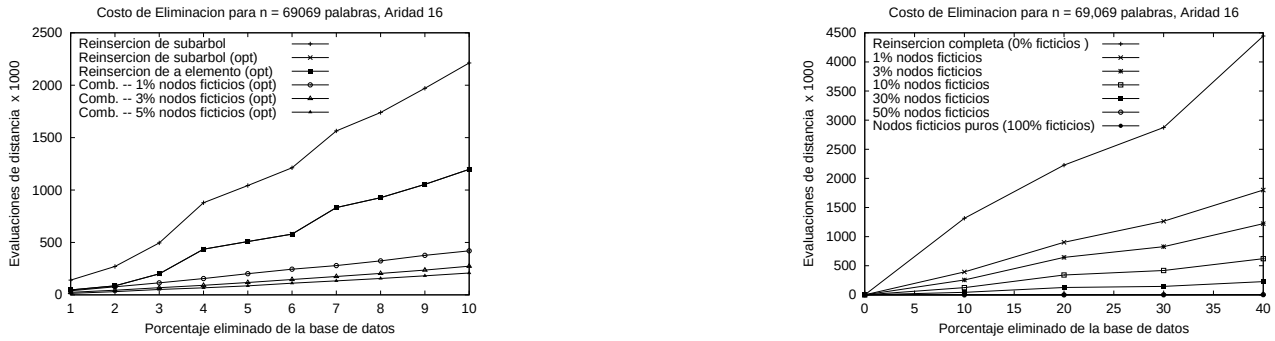

Figura 5: Costos de eliminación usando los diferentes métodos.

Consideremos ahora cómo el costo de búsqueda es afectado por las eliminaciones. Buscamos sobre un índice que contiene la mitad de los elementos de la base de datos. Esta mitad se construye insertando más elementos y luego eliminando suficientes elementos para dejar el $50\,\%$ del conjunto en el índice. Así, comparamos la búsqueda sobre conjuntos del mismo tamaño donde un porcentaje de los elementos se ha eliminado para dejar el conjunto de ese tamaño. Por ejemplo, $30\,\%$ de eliminaciones significa que se insertaron 49335 elementos y luego se eliminaron 14801, de manera tal de dejar 34534 elementos (la mitad del conjunto).

La Figura 6 muestra los resultados. Como se puede ver aún con reinserción completa ($\alpha = 0\,\%$) la calidad de la búsqueda se degrada, aunque apenas notablemente y no monotónicamente con el número de eliminaciones realizadas. A medida que α crece, los costos de búsqueda se incrementan debido a la necesidad de entrar en cada hijo de los nodos ficticios. La diferencia en costos de búsqueda deja de ser razonable tan pronto como $\alpha = 10\,\%$, y de hecho es significativo aún para $\alpha = 1\,\%$. Así uno tiene que elegir el correcto balance entre costos de eliminación y búsqueda dependiendo de la aplicación. Un buen balance para strings es $\alpha = 1\,\%$.

La Figura 7 muestra los datos para permitir comparar el cambio en los costos de búsqueda a medida que crece el α para el espacio de las palabras (las gráficas de arriba) y para el espacio de vectores de dimensión 101 con distribución de Gauss (las gráficas de abajo).

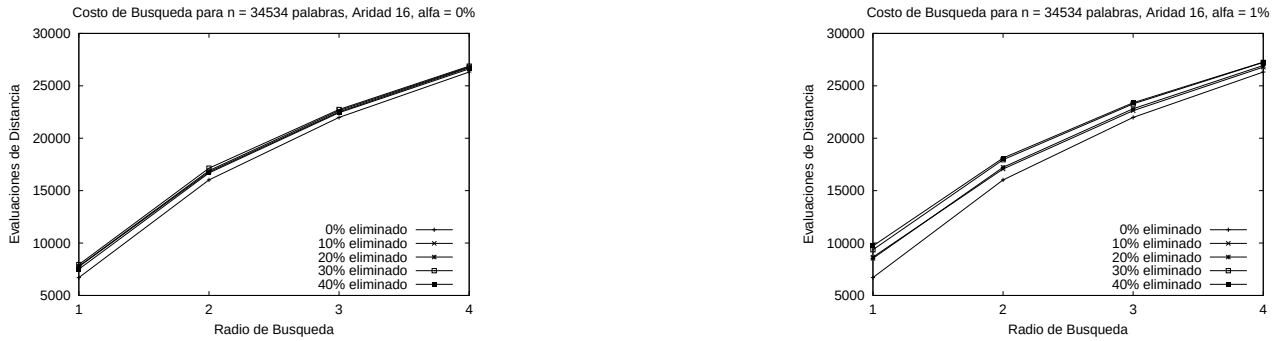

Figura 6: Costos de búsqueda usando diferentes métodos de eliminación. A la izquierda el caso de $\alpha = 0\,\%$, y a la derecha de $1\,\%$.

5. Conclusiones

Hemos presentado una versión dinámica de la estructura de datos *sa-tree*, la cual es capaz de realizar inserciones y eliminaciones eficientemente sin afectar su calidad de búsqueda. Muy pocas estructuras de datos para buscar en espacios métricos son completamente dinámicas, y en particular muy pocas son capaces de eliminar elementos. Además, ya hemos mostrado que mejoramos el comportamiento del *sa-tree* en espacios de baja dimensión, tanto para los costos de construcción y búsqueda.

El *sa-tree* original era una estructura de datos prometedora para búsqueda en espacios métricos, con varias desventajas que la hacían poco práctica: alto costo de construcción en espacios de baja dimensión, pobre desempeño de las búsquedas en espacios de baja dimensión o consultas con baja selectividad, y no ser capaz de realizar inserciones y eliminaciones.

Hemos salvado todas estas debilidades. Nuestro nuevo *sa-tree* dinámico se presenta como una estructura de datos práctica y eficiente que se puede utilizar en un amplio rango de aplicaciones, mientras mantiene las buenas características de la estructura de datos original.

Como un ejemplo para dar una idea del comportamiento de nuestro nuevo *sa-tree* dinámico, consideremos un espacio de vectores de dimensión 15 generados aleatoriamente con distribución uniforme en el cubo real unitario y usando aridad 16. Ahorramos el 52.63 % del costo de construcción estática, y mejoramos en promedio el tiempo de búsqueda por 0.91 %. Una eliminación con reinserción completa de elementos cuesta en promedio 143 evaluaciones de distancia, lo cual es 2.43 veces el costo de una inserción. Si permitimos 10 % de nodos ficticios en la estructura, entonces el costo de una eliminación baja a 17 y el costo de búsqueda se vuelve 3.04 % peor que la versión estática.

Además en el espacio de vectores de dimensión 101 con distribución de Gauss (un espacio de baja dimensión) y aridad 4, se obtienen mejoras significativas en la construcción y en las búsquedas, tanto antes como después de realizar eliminaciones. Los costos en este espacio usando *sa-tree* dinámico representan aproximadamente un 20 % de los obtenidos con el estático. En este espacio es interesante notar no sólo las mejoras obtenidas respecto de la versión estática, sino también que en él, cuando varían los porcentajes eliminados, cambia radicalmente cuál valor de α es más conveniente. Más adelante nos proponemos analizar la razón.

Actualmente estamos trabajando en hacer que el *sa-tree* trabaje eficientemente en memoria secundaria. En ese caso serán relevantes tanto el número de evaluaciones de distancia como el de accesos a disco.

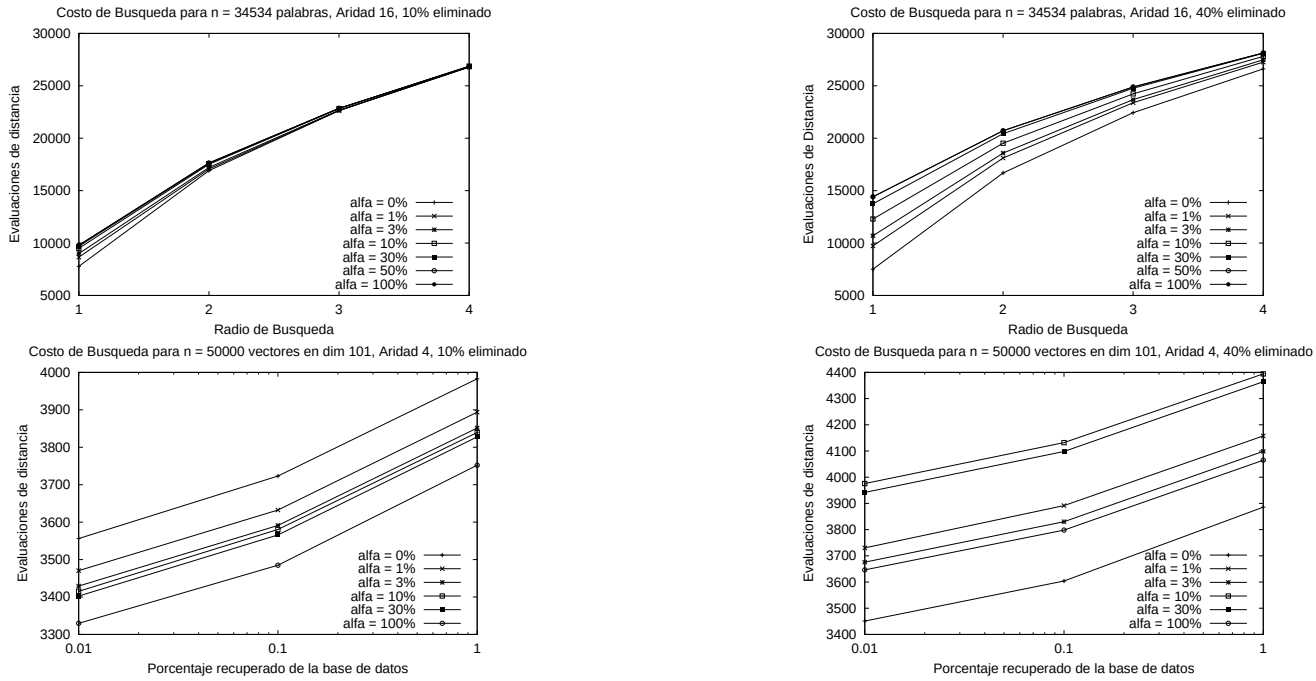

Figura 7: Costos de búsqueda usando diferentes métodos de eliminación, comparando α. Sobre la izquierda eliminamos el 10 % de la base de datos, sobre la derecha el 40 %.

Referencias

[1] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[2] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[3] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[5] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

[6] G. Navarro and N. Reyes. Dynamic spatial approximation trees. In *Proc. XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 213–222. IEEE CS Press, 2001.

[7] G. Navarro and N. Reyes. Improved dynamic spatial approximation trees. 2002. Manuscrito enviado a CLEI'2002 para su evaluación.

[8] Gonzalo Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.